Solutions for CP Exercises, October 2

1. Solution for Andrews Ex. 5.8

(a) The required invariant must state that the balance never becomes negative:

$$I \triangleq Bal \geq 0$$

The basic problem in this exercise is that the waiting condition for each *withdraw*(*amount*) operation depends on the parameter value *amount*. A general solution to this is to use a *covering condition*, i.e. to wake up all waiting processes, whenever the balance has improved. It is assumed that all amounts belongs to a type of positive integers *PosInt*.

monitor *SimpleAccount*

  var $Bal : integer := 0;$
    $positive : condition;$

  procedure $deposit(amount : PosInt)$
    $Bal := Bal + amount;$
    $signal\_all(positive);$

  procedure $withdraw(amount : PosInt)$
    while $Bal < amount$ **do** $wait(positive);$
    $Bal := Bal - amount;$

end

(b) Under the standard assumption the the condition queues are FiFO, the customers may be served FCFS by waking only one at a time, but only as long as the balance is large enough (using the magic *amount* function). Special care must be taken to prevent outside processes from making withdrawals before the woken processes. This could be accomplished by letting the deposit operation do the balance decrementation as in the FIFO Semaphore solution shown in Andrews Figure 5.3. Here, however, we take a more general approach. Whenever a withdrawal process is woken, the monitor is considered *busy*, and new processes will have to wait. Now the processes are started in FIFO order by a cascade wakeup:

monitor *MagicFSCSAccount*

  var $Bal : integer := 0;$
    $Busy : boolean := false;$
    $positive : condition;$

  procedure $deposit(amount : PosInt)$
    $Bal := Bal + amount;$
    if $\neg Busy \wedge \neg empty(positive) \wedge amount(positive) \leq Bal$ **then**
      $Busy := true;$
      $signal(positive);$

  procedure $withdraw(amount : PosInt)$
    if $Busy \vee Bal < amount$ **then**
      $wait(positive);$

$\quad\quad Busy := false;$
$\quad Bal := Bal - amount;$ — Bal assumed large enough
$\quad$ **if** $\neg Busy \land \neg empty(positive) \land amount(positive) \leq Bal$ **then**
$\quad\quad Busy := true;$
$\quad\quad signal(positive);$

end

Note that deposit processes may increment Bal, even when the monitor is busy, but that will not violate the expectations of the woken withdrawal process.

(c) In order to implement the magic function *amount* giving the requested amount of the first withdrawal process, two ideas may be applied:

- A new, separate condition queue is used by the first waiting process and that processes may set a global amount variable. Further processes will have to wait on the old queue. Now great care must be taken to ensure that exactly one and only one of the waiting processes proceed to this queue.
- Within the monitor, a datastructure is maintained giving the amounts of the waiting processes. Thus the datastructure will be parallel to the condition queue. Since the queue is supposed to be FIFO, a list type will be appropiate.

Here we choose the latter approach, extending the above solution with a list type with operations *append*, *head* and *tail*:

monitor *FSCSAccount*

var $Bal : integer := 0;$
$\quad Busy : boolean := false;$
$\quad amounts : List$ **of** $integer;$
$\quad positive : condition;$

procedure $deposit(amount : PosInt)$
$\quad Bal := Bal + amount;$
$\quad$ **if** $\neg Busy \land \neg empty(positive) \land head(amounts) \leq Bal$ **then**
$\quad\quad Busy := true;$
$\quad\quad amounts := tail(amounts);$
$\quad\quad signal(positive);$

procedure $withdraw(amount : PosInt)$
$\quad$ **if** $Busy \lor Bal < amount$ **then**
$\quad\quad amounts := append(amounts, amount);$
$\quad\quad wait(positive);$
$\quad\quad Busy := false;$
$\quad Bal := Bal - amount;$ — Bal assumed large enough
$\quad$ **if** $\neg Busy \land \neg empty(positive) \land head(amounts) \leq Bal$ **then**
$\quad\quad Busy := true;$
$\quad\quad amounts := tail(amounts);$
$\quad\quad signal(positive);$

end

2. Solution for Andrews Ex. 5.13

Let ns, ni, and nd denote the number of active searchers, inserters, and deleters respectively. Then the required invariant is:

$$I \triangleq 0 \le ns \land 0 \le ni \le 1 \land 0 \le nd \le 1 \land (nd > 0 \Rightarrow ns = 0 \land ni = 0)$$

Implementing these figures by monitor variables, the monitor is readily implemented:

```
monitor ListControl

  var ns, ni, nd : integer := 0;
      ok_search : condition;
      ok_insert : condition;
      ok_delete : condition;

  procedure start_search()
    while nd > 0 do wait(ok_search);
    ns := ns + 1;

  procedure end_search()
    ns := ns − 1;
    if ns = 0 ∧ ni = 0 then signal(ok_delete)

  procedure start_insert()
    while nd > 0 ∨ ni > 0 do wait(ok_insert);
    ni := ni + 1;

  procedure end_insert()
    ni := ni − 1;
    if ns = 0 ∧ empty(ok_insert) then signal(ok_delete)
                                 else signal(ok_insert)

  procedure start_delete()
    while ns > 0 ∨ ni > 0 ∨ nd > 0 do wait(ok_delete);
    nd := nd + 1;

  procedure end_delete()
    nd := nd − 1;
    if empty(ok_search) ∧ empty(ok_insert) then signal(ok_delete)
                                           else signal(ok_insert)
                                                signal_all(ok_search)

end
```

Here, it has been chosen to give preference to searchers/inserters. Another strategy could be to give preference to deleters or to alternate between the these two groups.

3. Solution for Andrews Ex. 5.11

Signal-and-continue semantics:

```
monitor Swap

  var val1, val2 : integer;
      swapping : boolean := false;
      swap : condition;                  — First waits here
      done : condition;                  — Wait for swapping to end

  procedure exchange(var value : integer)
    while swapping do wait(done);
    if empty(swap) then { val1 := value;
                          wait(swap);
                          value := val2;
                          swapping := false;
                          signal(done);  signal(done) }
                   else { val2 := value;
                          value := val1;
                          swapping := true;
                          signal(swap) }

end
```

When two processes are ready to exchange values, new processes must not interfere. This is ensured by the flag *swapping* that blocks new processes on the *done* queue. When the exchange is over, new processes may enter. However at most two of them can engage in a new exchange (they may be overtaken by other processes though, and thus a **while**-test is still needed).

[Due to the unconditional waits, this solutions is **not** robust towards spurious wakeups.]

Signal-and-urgent-wait semantics:

```
monitor Swap

  var val1, val2 : integer;
      swap : condition;                  — First waits here

  procedure exchange(var value : integer)
    if empty(swap) then { wait(swap);
                          val1 := value;
                          value := val2 }
                   else { val2 := value;
                          signal(swap);
                          value := val1 }

end
```

With this semantics, the monitor is greatly simplified by the fact that no new processes will interfere. The *signal*(*swap*) acts like a procedure call that activates the process waiting at *swap*.